Consider Rust in 2024

Nebojsa Koturovic

May 29, 2024 - 15:30

Abstract

Why I think Rust might be worth it in 2024

Contents

Rust in 2024

The article is based on my personal opinions, and it's mostly subjective, so I free myself from proving anything I say.

C++, Go and Rust

I'm interested in `C++`, `Go` and `Rust`, and I find them equally great options in 2024. In this article, my main focus will be on Rust, not because it's my favorite (it isn't), but mostly because I think it is worth learning, and a safer bet to learn in 2024 than in the years before.

First look

My Rust journey began about three years ago, I don't remember how I stumbled upon it exactly, it's probably due to its advertisement as a new shiny systems programming language. I do remember, however, that it didn't leave a strong impression on me. By just looking into the language, I couldn't say much, it seemed like an odd mixture of C++ and Java with a functional touch.

If you're a genuine C++ developer, like I was at the time, hearing the word macro would cause strong symptoms of nausea. That is exactly how I reacted the first time when I found out that `println!` is a macro.

```rust
fn main() {
    println!("Hello World!");
}
```

Even in the most basic examples of a program, macros come into play. As a C++ developer, my instinctive reaction was to close the browser tab and move on with my life, that certainly doesn't sound revolutionary, Rust must suck!

Second look

The time when I took a deeper look and realized that Rust might not be that bad was in the Spring of 2020, during the COVID-19 lockdown. I was amazed at code generation capabilities, after some more digging, I started to appreciate things like linear types, borrow checker, and user-friendly ecosystem.

Serialization/Deserialization in Rust

```rust
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    // Define an immutable point variable
    let point = Point { x: 1, y: 2 };

    // Convert the point to string and print it
    let serialized = serde_json::to_string(&point).unwrap();
    println!("serialized = {}", serialized);

    // Convert the string back to a Point and debug print it
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = {:?}", deserialized);
}
```

Output:

```
serialized = {"x":1,"y":2}
deserialized = Point { x: 1, y: 2 }
```

I challenge you to write an equivalent piece of code in `C++17`, or `C++20`.

This example takes a special place in my heart, it was amazing to see `json` serialization and deserialization working out of the box. I went through the process of making it possible in my cpp-rest-server project, and it was a very painful experience. I had to use a third-party static reflection library and write a custom serialization/deserialization logic to make it happen.

Jeff Atwood: "The best code is no code at all.

Why Rust?

In my opinion, Rust isn't primarily successful because of safety, or shiny new features. The initial success came from the fact that the centralized package model and good tooling around it made it super easy to get started and leverage third-party packages. Of course, there was a big dissatisfaction in the slow-moving C++ community, no default way to do things, and so on.

Nowadays, Rust is becoming widely used, accepted, and mature enough, which I think makes it a valuable option.

Don't get me wrong, I think there are great alternatives, take Go for example. I think Go is a great language, it has amazing tooling, simple to get started and build powerful software. I see the benefits of having a simpler language. However, such languages can be a tiny bit more verbose in my opinion.

Go and Rust have different philosophies, Rust embraces zero-cost abstractions, it's not garbage collected, and has a very strict and strong type system, whereas Go gives a bit more flexibility to its users. Ultimately, they have different use cases, but substantial overlap as well, personal preference can have an important role in picking one over another too.

Reasons to consider Rust

1. **Maturity** - In 2024, we can say that Rust has become widely accepted and mature enough
2. **Performance** - Rust offers performance equivalent to other compiled languages
3. **Type system** - Rust has a very strict and strong type system, which makes it a great choice for large projects
4. **Safety** - Eliminating a large portion of bugs, now enforced with Rust's type system

5. **Easy start** - Even tho Rust is a relatively complex language, it's pretty easy to get started

Apart from the given reasons, in the last year some major players like Microsoft, and Linux Foundation started adopting Rust. Adoptions and backing-up from such organizations provide more confidence in choosing Rust as major organizations start relying on it.

Reasons to avoid Rust

1. **Complexity** - Rust can be very complex at times, especially when dealing with lifetimes and macros
2. **Unsafe Rust** - Unsafe Rust can be very challenging
3. **Colored functions** - The existence of async and lifestyle annotated functions adds to the complexity
4. **C interop** - I don't find `C` interop particularly great, I would argue that `C++` is much better at this

Most of the downsides come as a result of incorporating safety into the language. Another downside that is not covered here is the Job market, most of the roles are around the crypto ecosystem.

Conclusion

Rust certainly looks like a compelling option in 2024, especially as it becomes adopted by large organizations. A strong type system, amazing tooling, and a fast-growing community make it a good choice both for new and experienced developers.

I started learning Rust to enrich my Programming skills and learn new programming concepts like lifetimes. It can be considered a relatively high-level language, so it's relatively easy to get work done.

If you're interested, you can take a look at my first-ever Rust project, it's a small program that simulates slerp animation. It was a great tool for the job, as it made very simple interface and abstractions to draw shapes and use a built-in camera: simple-slerp-project